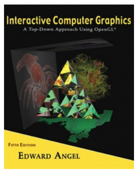

Interactive Computer Graphics
A Top-Down Approach Using OpenGL
Fifth Edition
Edward Angel

Name: ______________________ Term # ____

Homework 7 SOLUTIONS

(400 points)

NOTE: **Chapter 7 of the textbook shows the implementation steps for clipping, rasterization, and hidden-surface removal.**
Part A is intended to be done by hand.
Part B is an OpenGL application.

A. **(300 pts) Paper and Pencil**

***(Guidelines:** Read the material from the textbook chapter, you can use textbook figures to exemplify your answer, use keywords, summarize your answer, but the answer **cannot be longer the 7 lines!**)*

7.1 BASIC IMPLEMENTATION STRATEGIES

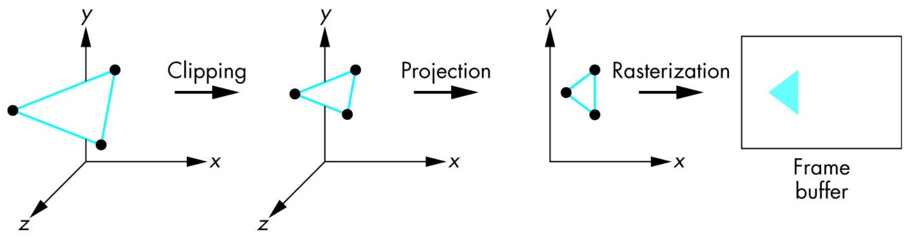
a. Explain Object oriented approach implementation.

ANSWER:

(Angel pp. 330)

Lets begin with a high- level view of the implementation process. In computer graph-ics, we start with an application program, and we end with an image. We can again consider this process as a black box (Figure 7.1) whose inputs are the vertices and states defined in the program geometric objects, attributes, camera specifications and whose output is an array of colored pixels in the frame buffer. Within the black box, we must do many tasks, including transformations, clip-ping, shading, hidden- surface removal, and rasterization of the primitives that can appear on the display. These tasks can be organized in a variety of ways, but regard-less of the strategy that we adopt, we must always do two things: We must pass every geometric object through the system, and we must assign a color to every pixel in the color buffer that is displayed. Suppose that we think of what goes into the black box in terms of a single program that carries out the entire process. This program takes as input sets of vertices specifying geometric objects and produces as output pixels in the frame buffer. Because this program must assign a value to every pixel and must process every geometric primitive (and every light source), we expect this programto contain at least two loops that iterate over these basic variables.

If we wish to write such a program, then we must immediately address the following question: Which variable controls the outer loop? The answer we choose determines the flow of the entire implementation process. There are two fundamental answers. The two strategies that follow are often called the image- oriented and the object- oriented approaches.



In the object-oriented approach, the outer loop is over the objects. We can think of the program as controlled by a loop of the following form: for(each_ object) render(object); A pipeline renderer fits this description. Vertices are defined by the program and flow through a sequence of modules that transforms them, colors them, and deter-mines whether they are visible. A polygon might flow through the steps illustrated in Figure 7.2. Note that after a polygon passes through geometric processing, the rasterization of this polygon can potentially affect any pixel in the frame buffer. Most implementations that follow this approach are based on construction of a rendering pipeline that contains hardware or software modules for each of the tasks. Data (vertices) flow forward through the system. In the past, the major limitations of the object- oriented approach were the large amount of memory required and the high cost of processing each object independently. Any geometric primitive that emerges from the geometric processing potentially can affect any set of pixels in the frame buffer; thus, the entire color buffer and various other buffers, such as the depth buffer used for hidden- surface removal must be of the size of the display and must be available at all times. Before memory became inexpensive and dense, this requirement was considered to be a serious problem. Now various pipelined geometric processors are available that can process tens of millions of polygons per second. In fact, precisely because we are doing the same operations on every primitive, the hardware to build an object-based system is fast and relatively inexpensive, with many of the functions implemented with special- purpose chips. Today, the main limitation of object- oriented implementations is that they can-not handle most global calculations. Because each geometric primitive is processed independently and in an arbitrary order complex shading effects that involve multiple geometric objects, such as reflections, cannot be handled, except by approximate methods. The major exception is hidden- surface removal, where the z- buffer is used to store global information.

b. Explain Image oriented approach implementation.
ANSWER:
(Angel pp. 331)
Image- oriented approaches loop over pixels, or rows of pixels called scanlines, that constitute the frame buffer. In pseudocode, the outer loop of such a program is of the following form: for(each_ pixel) assign_ a_ color(pixel); For each pixel, we work backward, determining which geometric primitives can contribute to its color. The advantages of this approach are that we need only limited display memory at any time and that we can hope to generate pixels at the rate and in the order required to refresh the display. Because the results of most calculations do not differ greatly from pixel to pixel (or scanline to scanline), we can use this coherence in our algorithms by developing incremental forms for many of the steps in the implementation. The main disadvantage of this approach is that unless we first build a data structure from the geometric data, we do not know which primitives affect which pixels. Such a data structure can be complex and may imply that all the geometric data must be available at all times during the rendering process. For problems with very large databases, even having a good data representation may not to all objects for each pixel, they are well suited to handle global effects such as shadows and reflections. Ray tracing (Chapter 13) is an example of the image- based approach. Because our primary interest is in interactive applications, we lean toward the object- based approach, although we look at examples of algorithms suited for both approaches. In addition, as hardware improves in both speed and lowered cost, approaches that are not competitive today may well be practical in the near future.

7.3 CLIPPING

a. Explain when is clipping done in OpenGL and why.
ANSWER:
(Angel pp. 336)
We can now turn to clipping, the process of determining which primitives, or parts of primitives, fit within the clipping or view volume defined by the application program. Clipping is done before the perspective division that is necessary if the w component of a clipped vertex is not equal to 1. The portions of all primitives that can possibly be displayed we have yet to apply hidden- surface removal lie within the cube as follows: - w = x = w, - w = y = w, - w = z = w. This coordinate system is called clip coordinates, and it depends on neither the original application units nor the particulars of the display

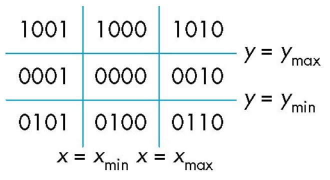
device, although the information to produce the correct image is retained in this coordinate system. Note also that projection has been carried out only partially. We still must do the perspective division and the final orthographic projection. After perspective division, we have a three- dimensional representation in normalized device coordinates. By carrying our clipping in clip coordinates, we avoid doing the perspective division for primitives that lie outside the clipping volume. We will concentrate on clipping of line segments and polygons because they are the most common primitives to pass down the pipeline. Although the OpenGL pipeline does clipping on three- dimensional objects, there are other systems in which the objects are first projected into the x, y plane. Fortunately, many of the most efficient algorithms are almost identical in two and three dimensions and we will focus on these algorithms.

7.4 LINE-SEGMENT CLIPPING

a. Explain culled.
ANSWER:
(Angel pp. 336)
A clipper decides which primitives, or parts of primitives, can possibly appear on the display and are passed on to the rasterizer. Primitives that fit within the specified view volume pass through the clipper, or are accepted. Primitives that cannot appear on the display are eliminated, or rejected or culled. Primitives that are only partially within the view volume must be clipped such that any part lying outside the volume is removed. Clipping can occur at one or more places in the viewing pipeline. The modeler may clip to limit the primitives that the hardware must handle. The primitives may be clipped after they have been projected from three- to two- dimensional objects. In OpenGL, primitives are clipped against a three- dimensional view volume before rasterization. We will develop a sequence of clippers. For both pedagogic and historic reasons, we start with two two- dimensional line-segment clippers. Both extend directly to three dimensions and to clipping of polygons.

7.4.1 Cohen-Sutherland Clipping

a. Explain Figure 7.6.
ANSWER:
(Angel pp. 337)



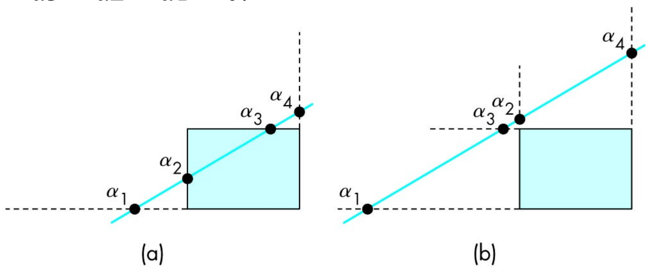
7.4.1 Cohen- Sutherland Clipping The two- dimensional clipping problem for line segments is shown in Figure 7.5. We can assume for now that this problem arises after three- dimensional line segments have been projected onto the projection plane, and that the window is part of the projection plane that is mapped to the viewport on the display. All values are specified as real numbers. We can see that the entire line segment AB appears on the display, whereas none of CD appears. EF and GH have to be shortened before being displayed. Although a line segment is completely determined by its endpoints, GH shows that, even if both endpoints lie outside the clipping window, part of the line segment may still appear on the display. We could compute the intersections of the lines of which the segments are parts with the sides of the window, and thus could determine the necessary information for clipping. However, we want to avoid intersection calculations, if possible, because each intersection requires a floating- point division. The Cohen- Sutherland algorithm was the first to seek to replace most of the expensive floating- point multiplications and divisions with a combination of floating- point subtractions and bit operations. The algorithm starts by extending the sides of the window to infinity, thus breaking up space into the nine regions shown in Figure 7.6. Each region can be assigned a unique 4- bit binary number, or outcode, b0b1b2b3, as follows. Suppose that (x, y) is a point in the region; then, Likewise, b1 is 1 if y < ymin, and b2 and b3 are determined by the relationship between x and the left and right sides of the window. The resulting codes are indicated in Figure 7.7. For each endpoint of a line segment, we first compute the endpoints outcode, a step that can require eight floating-point subtractions per line segment. Consider a line segment whose outcodes are given by o1 = outcode(x1, y1) and o2 = outcode(x2, y2). We can now reason on the basis of these outcodes.

b. Explain the four cases for the endpoint outcodes.
ANSWER:
(Angel pp. 337-338)
There are four cases: 1. (o1 = o2 = 0.) Both endpoints are inside the clipping window, as is true for 2. (o1 = 0, o2 = 0; or vice versa.) One endpoint is inside the clipping window; one is outside (see segment CD in Figure 7.7). The line segment must be shortened. The nonzero outcode indicates which edge, or edges, of the window are crossed by the segment. One or two intersections must be computed. Note that after one intersection is computed, we can compute the outcode of the point of intersection to determine whether another intersection calculation is required. 3. (o1 & o2 = 0.) By taking the bitwise and of the outcodes, we determine whether or not the two endpoints lie on the same outside side of the window. If so, the line segment can be discarded (see segment EF in Figure 7.7). 4. (o1 & o2 = 0.) Both endpoints are outside, but they are on the outside of different edges of the window. As we can see from segments GH and IJ in Figure 7.7, we cannot tell from just the outcodes whether the segment can be discarded or must be shortened. The best we can do is to intersect with one of the sides of the window, and to check the outcode of the resulting point.

7.4.2 Liang-Barsky Clipping

a. Explain Figure 7.8.
ANSWER:
(Angel pp. 340)
7.4.2 Liang- Barsky Clipping If we use the parametric form for lines, we can approach the clipping of line segments in a different and ultimately more efficient manner. Suppose that we have a line segment defined by the two endpoints p1 = [x1, y1] T and p2 = [x2, y2] T. We can use these endpoints to define a unique line that we can express

parametrically, either in matrix form, p(a) = (1- a) p1 + ap2, or as two scalar equations, x(a) = (1- a) x1 + ax2, y(a) = (1- a) y1 + ay2. Note that this form is robust and needs no changes for horizontal or vertical lines. As the parameter a varies from 0 to 1, we move along the segment from p1 to p2. Negative values of a yield points on the line on the other side of p1 from p2. Similarly, values of a > 1 give points on the line past p2 going off to infinity. Consider the line segment and the line of which it is part, as shown in Figure 7.8(a). As long as the line is not parallel to a side of the window (if it is, we can handle that situation with ease), there are four points where the line intersects the extended sides of the window. These points correspond to the four values of the parameter: a1, a2, a3, and a4. One of these values corresponds to the line entering the window; another corresponds to the line leaving the window. Leaving aside, for the moment, how we compute these intersections, we can order them, and can determine which correspond to intersections that we need for clipping. For the given example, 1> a4 > a3 > a2 > a1 > 0.



Hence, all four intersections are inside the original line segment, with the two innermost (a2 and a3) determining the clipped line segment. We can distinguish this case from the case in Figure 7.8(b), which also has the four intersections between the endpoints of the line segment, by noting that the order for this case is 1> a4 > a2 > a3 > a1 > 0. The line intersects both the top and the bottom of the window before it intersects either the left or the right; thus, the entire line segment must be rejected. Other cases of the ordering of the points of intersection can be argued in a similar way. Efficient implementation of this strategy requires that we avoid computing inter-sections until they are needed. Many lines can be rejected before all four intersections are known. We also want to avoid floating- point divisions where possible. If we use the parametric form to determine the intersection with the top of the window, we find the intersection at the value a = ymax - y1 y2 - y1 . Similar equations hold for the other three sides of the window. Rather than computing these intersections, at the cost of a division for each, instead we write the equation as a(y2 - y1) = a y = ymax - y1 = ymax.

All the tests required by the algorithm can be restated in terms of ymax, y, and similar terms can be computed for the other sides of the windows. Thus, all decisions about clipping can be made without floating- point division. Only if an intersection is needed (because a segment has to be shortened) is the division done. The efficiency of this approach, compared to that of the Cohen- Sutherland algorithm, is that we avoid multiple shortening of line segments and the related reexecutions of the clipping algorithm. We forgo discussion of other efficient two- dimensional line- clipping algorithms because, unlike the Cohen- Sutherland and Liang- Barsky algorithms, these algorithms do not extend to three dimensions.

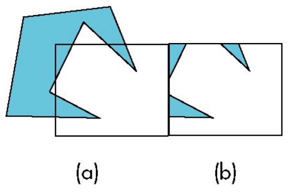
7.5 POLYGON CLIPPING

a. Explain Figure 7.10

ANSWER:

(Angel pp. 341)

7.5 POLYGON CLIPPING Polygon clipping arises in a number of ways. Certainly, we want to be able to clip polygons against rectangular windows for display. However, at times we may want windows that are not rectangular. Other parts of an implementation, such as shadow generation and hidden- surface removal, can require clipping of polygons against other polygons. For example, Figure 7.9 shows the shadow of a polygon that we create by clipping a polygon that is closer to the light source against polygons that are farther away. Many antialiasing methods rely on our ability to clip polygons against other polygons. We can generate polygon- clipping algorithms directly from line- clipping algorithms by clipping the edges of the polygon successively. However, we must be careful to remember that a polygon is a two- dimensional object with an interior, and depending on the form of the polygon, we can generate more than one polygonal object by clipping. Consider the nonconvex (or concave) polygon shown in Figure 7.10(a). If we clip it against a rectangular window, we get the result shown in Figure 7.10(b). Most viewers looking at this figure would conclude that we have generated three polygons by clipping. Unfortunately, implementing a clipper that can increase the number of objects can be a problem. We could treat the result of the clipper as a single polygon, as shown in Figure 7.11, with edges that overlap along the sides of the window, but this choice might cause difficulties in other parts of the implementation. Convex polygons do not present such problems. Clipping a convex polygon against a rectangular window can leave at most a single convex polygon (see Exercise 7.3). A graphics system might then either forbid the use of concave polygons, or divide (tessellate)



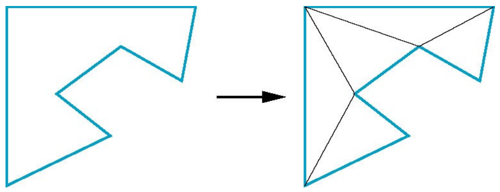
b. Explain Sutherland-Hodgeman polygon clipping

ANSWER:

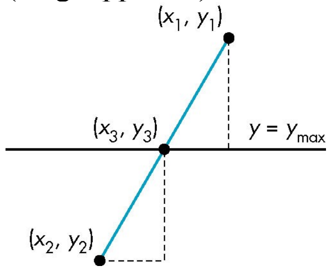
(Angel pp. 341)

For rectangular clipping regions, both the Cohen- Sutherland and the Liang- Barsky algorithms can be applied to polygons on an edge- by- edge basis. There is another approach, developed by Sutherland and Hodgeman, that fits well with the pipeline architectures that we have discussed. A line- segment clipper can be envisioned as a black box (Figure 7.13) whose input is the pair of vertices from the segment to be tested and clipped, and whose output either is a pair of vertices corresponding to the clipped line segment or is nothing if the input line segment lies outside the window. Rather than considering the clipping window as four line segments, we can consider it as the object created by the intersection of four infinite lines that determine the top, bottom, right, and left sides of the window. We can then subdivide our clip-per into a pipeline of simpler clippers, each of which clips against a single line that is the extension of an edge of the window. We can use the black- box view on each of the individual clippers.

c. Explain Figure 7.12 tesselation
ANSWER:
(Angel pp. 341- 342)
A graphics system might then either forbid the use of concave polygons, or divide (tessellate)



d. Explain Figure 7.15 how are x3, y3 calculated
ANSWER:
(Angel pp. 342)

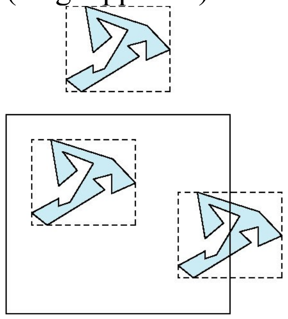


Suppose that we consider clipping against only the top of the window. We can think of this operation as a black box (Figure 7.14) whose input and output are pairs of vertices, with the value of ymax as a parameter known to the clipper. Using the similar triangles in Figure 7.15, we see that if there is an intersection, it lies at x3 = x1 + (ymax - y1) x2 - x1 y2 - y1 , y3 =

7.6 CLIPPING OF OTHER PRIMITIVES

7.6.1 Bounding Boxes and Volumes

a. Explain Figure 7.19
ANSWER:
(Angel pp. 343)



7.6.1 Bounding Boxes and Volumes Suppose that we have a many- sided polygon, as shown in Figure 7.18(a). We could apply one of our clipping algorithms, which would clip the polygon by individually clipping all that polygons edges. However, we can see that the entire polygon lies outside the clipping window. We can exploit this observation through the use of the axis- aligned bounding box or extent of the polygon (Figure 7.18(b)): the smallest rectangle, aligned with the window, that contains the polygon. Calculating the bounding box requires only going through the vertices of the polygon to find the minimum and maximum of both the x and y values. Once we have the bounding box, we can often avoid detailed clipping. Consider the three cases shown in Figure 7.19. For the polygon above the window, no clipping is necessary, because the minimum y for the bounding box is above the top of the window. For the polygon inside the window, we can determine that it is inside by comparing the bounding box with the window. Only when we discover that the bounding box straddles the window do we have to carry out detailed clipping, using all the edges of the polygon. The use of extents is such a powerful technique in both two and three dimensions that modeling systems often compute a bounding for each object, automatically, and store the bounding box with the object. Axis- aligned bounding boxes work in both two and three dimensions. In three dimensions, they can be used in the application to perform clipping to reduce the burden on the pipeline. Other volumes, such as spheres, can also work well. One of the other applications of bounding volumes is in collision detection (Chapter 12). One of the fundamental operations in animating computer games is to determine if two moving entities have collided. For example, consider two animated characters moving in a sequence of images. We need to know when they collide so that we can alter their paths. This problem has many similarities to the clipping problem because we want to determine when the volume of one intersects the volume of the other. The complexity of the objects and the need to do these calculations very quickly make this problem difficult. A common approach is to place each object in a bounding volume, either an axis- aligned bounding box or a sphere, and to determine if the volumes intersect. If they do, then detailed calculations can be done.

7.8 RASTERIZATION

a. Explain rasterization

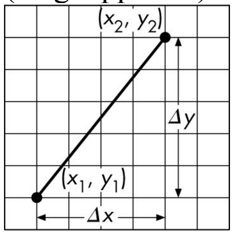
ANSWER:

(Angel pp. 349)

7.8 RASTERIZATION We are now ready to take the final step in the journey from the specification of geo-metric entities in an application program to the formation of fragments: rasterization of primitives. In this chapter, we are concerned with only line segments and polygons, both of which are defined by vertices. We can assume that we have clipped the primitives such that each remaining primitive is inside the view volume. Fragments are potential pixels. Each fragment has a location in screen coordinates that corresponds to a pixel location in the color buffer. Fragments can have other attributes, such as colors, which are computed by the rasterizer from vertex data. Thus, if shading is enabled the rasterizer gets vertex shades as input and interpolates a shade for each fragment it outputs. Fragments also carry depth information that can be used for hidden-surface removal. To clarify the discussion, we will ignore hidden- surface removal until Section 7.12 and thus we can work directly in screen coordinates. Because we are not considering hidden- surface removal, translucent fragments, or antialiasing, we can develop rasterization algorithms in terms of the pixels that they color. We further assume that the color buffer is an n m array of pixels, with (0, 0) corresponding to the lower- left corner. Pixels can be set to a given color by a single function inside the graphics implementation of the following form: write_ pixel(int ix, int iy, int value);

The argument value can be either an index, in color- index mode, or a pointer to an RGBA color. On the one hand, a color buffer is inherently discrete; it does not make sense to talk about pixels located at places other than integer values of ix and iy. On the other hand, screen coordinates, which range over the same values as do ix and iy, are real numbers. For example, we can compute a fragment location such as (63.4, 157.9) in screen coordinates, but we must realize that the nearest pixel is centered either at (63, 158) or at (63.5, 157.5), depending on whether pixels are considered to be centered at whole or half integer values. Pixels have attributes that are colors in the color buffer. Pixels can be displayed in multiple shapes and sizes that depend on the characteristics of the display. We address this matter in Section 7.13. For now, we can assume that a pixel is displayed as a square, whose center is at the location associated with the pixel and whose side is equal to the distance between pixels. In OpenGL, the centers of pixels are located at values halfway between integers. There are some advantages to this choice (see Exercise 7.19). We also assume that a concurrent process reads the contents of the color buffer and creates the display at the required rate. This assumption, which holds in many systems that have dual- ported memory, allows us to treat the rasterization process independently of the display of the contents of the frame buffer.

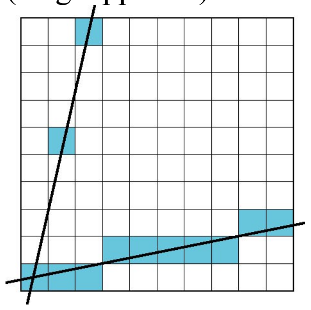
b. Explain Figure 7.27
ANSWER:
(Angel pp. 350)



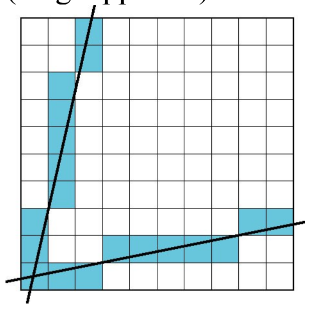
The simplest scan- conversion algorithm for line segments has become known as the DDA algorithm, after the digital differential analyzer, an early electromechanical device for digital simulation of differential equations. Because a line satisfies the differential equation dy dx = m, where m is the slope, generating a line segment is equivalent to solving a simple differential equation numerically. Suppose that we have a line segment defined by the endpoints (x1, y1) and (x2, y2). We assume that, because we are working in a color buffer, these values have been rounded to have integer values, so the line segment starts and ends at a known pixel. 2 The slope is given by m = y2 - y1 x2 - x1 = y x . We assume that 0 = m = 1. We can handle other values of m using symmetry. Our algorithm is based on writing a pixel for each value of ix in write_ pixel as x goes from x1 to x2. If we are on the line segment, as shown in Figure 7.27, for any change in x equal to x, the corresponding changes in y must be y = m x.
As we move from x1 to x2, we increase x by 1 each iteration; thus, we must increase y by y = m. Although each x is an integer, each y is not, because m is a floating- point number and we must round it to find the appropriate pixel, as shown in Figure 7.28. Our algorithm, in pseudocode, is for(ix= x1; ix <= x2; ix++) { y+= m; write_ pixel(x, round(y) , line_ color); }

where round is a function that rounds a real number to an integer. The reason that we limited the maximum slope to 1 can be seen from Figure 7.29. Our algorithm is of this form: For each x, find the best y. For large slopes, the separation between pixels that are colored can be large, generating an unacceptable approximation to the line segment. If, however, for slopes greater than 1, we swap the roles of x and y, the algorithm becomes this: For each y, find the best x. For the same line segments, we get the approximations in Figure 7.30. Note that the use of symmetry removes any potential problems from either vertical or horizontal line segments. You may want to derive the parts of the algorithm for negative slopes.

c. Explain Figure 7.29
ANSWER:
(Angel pp. 351)



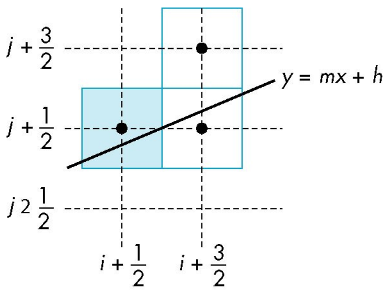
d. Explain Figure 7.30
ANSWER:
(Angel pp. 351)



Because line segments are determined by vertices, we can use interpolation to assign a different color to each pixel that we generate. We can also generate various dash and dot patterns by changing the color that we use as we generate pixels. Neither of these effects has much to do with the basic rasterization algorithm, as the latters job is to determine only which pixels to color, rather than to determine the color that is used.

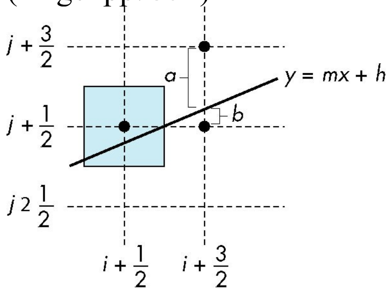
7.9 BRESENHAM'S ALGORITHM

a. Explain Figure 7.31
ANSWER:
(Angel pp. 352)



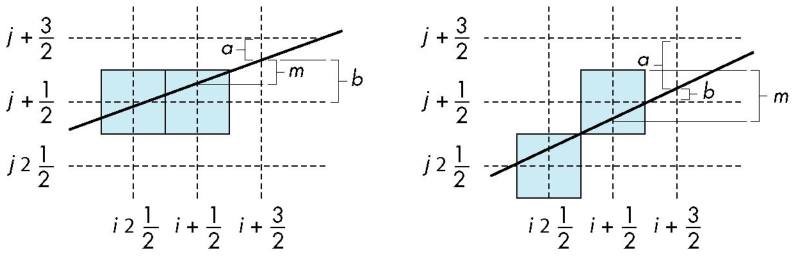
The DDA algorithm appears efficient. Certainly it can be coded easily, but it re-quires a floating- point addition for each pixel generated. Bresenham derived a line-rasterization algorithm that, remarkably, avoids all floating- point calculations and has become the standard algorithm used in hardware and software rasterizers. We assume, as we did with the DDA algorithm, that the line segment goes between the integer points (x1, y1) and (x2, y2), and that the slope satisfies 0 = m = 1. This slope condition is crucial for the algorithm, as we can see with the aid of Figure 7.31. Suppose that we are somewhere in the middle of the scan conversion of our line segment and have just placed a pixel at (i + 1 2 , j + 1 2). We know that the line of which the segment is part can be represented as y = mx + h. At x = i + 1 2 , this line must pass within one- half the length of the pixel at (i + 1 2 , j + 1 2); 3 otherwise, the rounding operation would not have generated this pixel. If we move ahead to x = i + 3 2 , the slope condition indicates that we must set the color of one of only two possible pixels: either the pixel at (i + 3 2 , j + 1 2), or the pixel at (i + 3 2 , j + 3 2). Having reduced our choices to two pixels, we can pose the problem anew in terms of the decision variable d = b - a, where a and b are the distances between the line and the upper and lower candidate pixels at x = i + 3 2 , as shown in Figure 7.32. If d is negative, the line passes closer to the lower pixel, so we choose the pixel at (i + 3 2 , j + 1 2); otherwise, we choose the pixel at (i + 3 2 , j + 3 2). Although we could compute d by computing y = mx + b, we hesitate to do so because m is a floating- point number.

a. Explain Figure 7.32
ANSWER:
(Angel pp. 352)



We obtain the computational advantages of Bresenhams algorithm through two further steps. First, we replace floating- point operations with fixed- point operations. Second, we apply the algorithm incrementally. We start by replacing d with the new decision variable d = (x2 - x1)(b - a) = x(b - a), a change that cannot affect which pixels are drawn, because it is only the sign of the decision variable that matters. If we substitute for a and b, using the equation of the line, and noting that m = y2 - y1 x2 - x1 = y x , h = y2 - mx2, then we can see that d is an integer. We have eliminated floating- point calculations, but the direct computation of d requires a fair amount of fixed- point arithmetic.

Lets take a slightly different approach. Suppose that dk is the value of d at x = k + 1 2 . We would like to compute dk+ 1, the value of the decision variable at x = k + 3 2 , incrementally from dk. There are two situations, depending on whether or not we increment the y location of the pixel at x = k + 1 2 ; these situations are shown in Figure 7.33. By observing that a is the distance between the location of the upper candidate location and the line, we see that a increases by m only if y was increased at the previous step; otherwise, it decreases by m - 1. Likewise, b either decreases by - m or increases by 1- m when we increment x. Multiplying by x, we find that the possible changes in d are either 2 y or 2(y - x). We can state this result in the iterative form dk+ 1 = dk + 2 y if dk < 0; 2(y - x) otherwise.



The calculation of each successive pixel in the color buffer requires only an addition and a sign test. This algorithm is so efficient that it has been incorporated as a single instruction on graphics chips. See Exercise 7.14 for calculation of the initial value d0.

7.10 POLYGON RASTERIZATION

a. Explain polygon rasterization problem

ANSWER:

(Angel pp. 354)

One of the major advantages that the first raster systems brought to users was the ability to display filled polygons. At that time, coloring each point in the interior of a polygon with a different shade was not possible in real time, and the phrases rasterizing polygons and polygon scan conversion came to mean filling a polygon with a single color. Unlike rasterization of lines, where a single algorithm dominates, there are many viable methods for rasterizing polygons. The choice depends heavily on the implementation architecture. We concentrate on methods that fit with our pipeline approach and can also support shading. In Sections 7.10.4 through 7.10.6, we survey a number of other approaches.

7.10 POLYGON RASTERIZATION

a. Explain polygon rasterization problem

ANSWER:

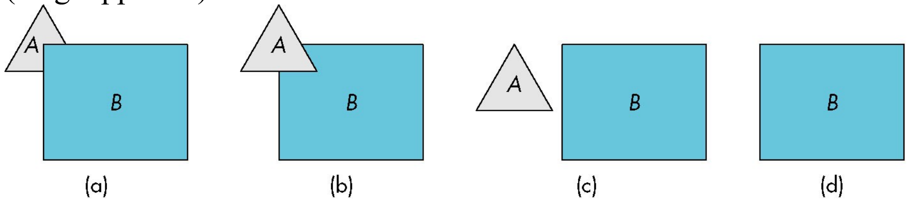
(Angel pp. 354)

7.11 HIDDEN-SURFACE REMOVAL

7.11.1 Object-Space and Image-Space Approaches

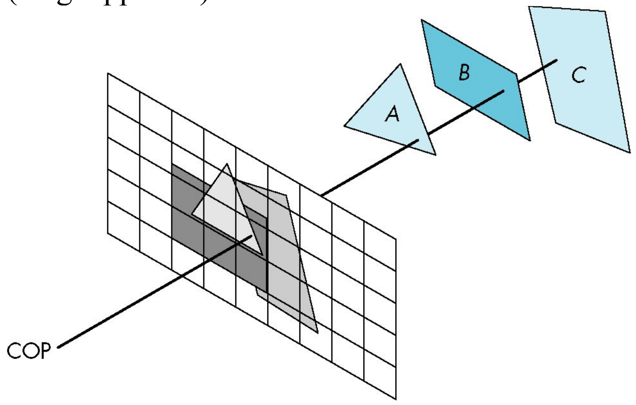
a. Explain Figure 7.39

ANSWER:
(Angel pp. 358)



The study of hidden- surface removal algorithms clearly illustrates the variety of available algorithms, the differences between working with objects and working with images, and the importance of evaluating the incremental effects of successive algorithms in the implementation process. Consider a scene composed of k three- dimensional opaque flat polygons, each of which we can consider to be an individual object. We can derive a generic object-space approach by considering the objects pairwise, as seen from the center of projection. Consider two such polygons, A and B. There are four possibilities (Figure 7.39):
1. A completely obscures B from the camera; we display only A. 2. B obscures A; we display only B. 3. A and B both are completely visible; we display both A and B. 4. A and B partially obscure each other; we must calculate the visible parts of each polygon. For complexity considerations, we can regard the determination of which case we have and any required calculation of the visible part of a polygon as a single operation. We proceed iteratively. We pick one of the k polygons and compare it pairwise with the remaining k - 1polygons. After this procedure, we know which part (if any) of this polygon is visible, and we render the visible part. We are now done with this polygon, so we repeat the process with any of the other k - 1 polygons. Each step involves comparing one polygon, pairwise, with the other remaining polygons, until we have only two polygons remaining, and we compare them to each other. We can easily determine that the complexity of this calculation is O(k2). Thus, without deriving any of the details of any particular object- space algorithm, we should suspect that the object- space approach works best for scenes that contain relatively few polygons.

b. Explain Figure 7.40
ANSWER:
(Angel pp. 359)



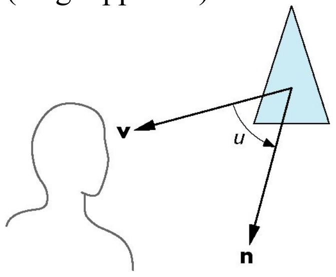
The image- space approach follows our viewing and ray- casting model, as shown in Figure 7.40. Consider a ray that leaves the center of projection and passes through a pixel. We can intersect this ray with each of the planes determined by our k polygons, determine for which planes the ray passes through a polygon, and finally, for those rays, find the intersection closest to the center of projection. We color this pixel with the shade of the polygon at the point of intersection. Our fundamental operation is the intersection of rays with polygons. For an n m display, we have to carry out this of the operations, we were able to get an upper bound. In general, the O(k) bound accounts for the dominance of image- space methods. The O(k) bound is a worst- case bound. In practice, image-space algorithms perform much better (see Exercise 7.9). However, because image- space approaches work at the fragment or pixel level, their accuracy is limited by the resolution of the frame buffer.

7.11.4 Back-Face Removal

a. Explain Figure 7.45 and culling

ANSWER:

(Angel pp. 361)



7.11.4 Back- Face Removal In Chapter 6, we noted that in OpenGL, we can choose to render only front- facing polygons. For situations where we cannot see back faces, such as scenes composed of convex polyhedra, we can reduce the work required for hidden-surface removal by eliminating all back- facing polygons before we apply any other hidden- surface removal algorithm. The test for culling a back- facing polygon can be derived from Figure 7.45. We see the front of a polygon if the normal, which comes out of the front face, is pointed toward the viewer. If . is the angle between the normal and the viewer, then the polygon is facing forward if and only if - 90 = . = 90, or, equivalently, cos . = 0. The second condition is much easier to test because, instead of computing the cosine, we can use the dot product: n . v >= 0.

We can simplify this test even further if we note that usually it is applied after transformation to normalized device coordinates. In this system, all views are orthographic, with the direction of projection along the z- axis. Hence, in normalized device coor-dinates, v =.. 0 0 1 .. . Thus, if the polygon is on the surface ax + by + cz + d = 0 in normalized device coordinates, we need only to check the sign of c to determine whether we have a front- or back- facing polygon. This test can be implemented easily in either hardware or software; we must simply be careful to ensure that removing back-facing polygons is correct for our application. In OpenGL, the function glCullFace allows us to turn on back- face elimination. OpenGL takes a different approach to back- face determination. The algorithm is based on computing the area of the polygon in screen coordinates. Consider the polygon shown in Figure 7.46 with n vertices. Its area is given by 1 2 i (yi+ 1 + yi)(xi+ 1 - xi),

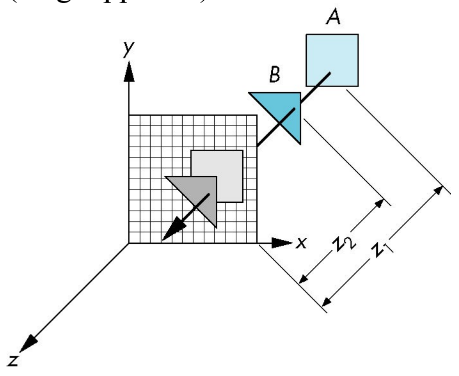
where the indices are taken modulo n (see Exercise 7.28). A negative area indicates a back- facing polygon.

7.11.5 The z-Buffer Algorithm

a. Explain Figure 7.47

ANSWER:

(Angel pp. 362)



The z- buffer algorithm is the most widely used hidden- surface removal algorithm. It has the advantages of being easy to implement, in either hardware or software, and of being compatible with pipeline architectures, where it can execute at the speed at which fragments are passing through the pipeline. Although the algorithm works in image space, it loops over the polygons, rather than over pixels, and can be regarded as part of the scan- conversion process that we discussed in Section 7.10. Suppose that we are in the process of rasterizing one of the two polygons shown in Figure 7.47. We can compute a color for each point of intersection between a ray from the center of projection and a pixel, using interpolated values of the vertex shades computed as in Chapter 6. In addition, we must check whether this point is visible. It will be visible if it is the closest point of intersection along the ray. Hence, if we are rasterizing B, its shade will appear on the screen if the distance z2 is less than the distance z1 to polygon A. Conversely, if we are rasterizing A, the pixel that corresponds to the point of intersection will not appear on the display. Because we are proceeding polygon by polygon, however, we do not have the information on all other polygons as we rasterize any given polygon. However, if we keep depth information with each fragment, then we can store and update depth information for each location in the frame buffer as fragments are processed. Suppose that we have a buffer, the z- buffer, with the same resolution as the frame buffer and with depth consistent with the resolution that we wish to use for distance. For example, if we have a 1024 1280 display and we use standard integers for the depth calculation, we can use a 1024 1280 z- buffer with 32- bit elements. Initially, each element in the depth buffer is initialized to a depth corresponding to the maximum distance away from the center of projection. 6 The color buffer is initialized to the background color. At any time during rasterization and fragment processing, each location in the z- buffer contains the distance along the ray corresponding to this location of the closest intersection point on any polygon found so far. The calculation proceeds as follows. We rasterize, polygon by polygon, using one of the methods from Section 7.10. For each fragment on the polygon corresponding to the intersection of the polygon with a ray through a pixel, we compute the depth from the center of projection. We compare this depth to the value in the z- buffer corresponding to this fragment. If this depth is greater than the depth in the z-

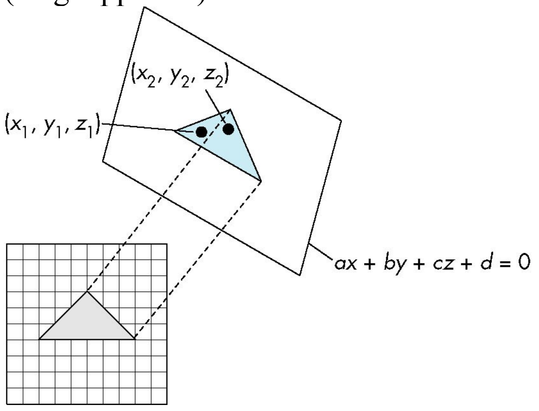
buffer, then we have already processed a polygon with a corresponding fragment closer to the viewer, and this fragment is not visible. If the depth is less than the depth in the z- buffer, 7 then we have found a fragment closer to the viewer. We up-date the depth in the z- buffer and place the shade computed for this fragment at the corresponding location in the color buffer. Note that for perspective views, the depth we are using in the z- buffer algorithm is the distance that has been altered by the normalization transformation that we discussed in Chapter 5. Although this trans-formation is nonlinear, it preserves relative distances. However, this nonlinearity can introduce numerical inaccuracies, especially when the distance to the near clipping plane is small. Unlike other aspects of rendering, where the particular implementation algorithms may be unknown to the user, for hidden- surface removal, OpenGL uses the z- buffer algorithm. This exception arises because the application program must initialize the z- buffer explicitly every time a new image is to be generated. The z- buffer algorithm works well with image- oriented approaches to implementation because the amount of incremental work is small. Suppose that we are rasterizing a polygon, scanline by scanline an option we examine in Section 7.9. The polygon is part of a plane (Figure 7.48) that can be represented as ax + by + cz + d = 0.
Suppose that (x1, y1, z1) and (x2, y2, z2) are two points on the polygon (and the plane). If x = x2 - x1, y = y2 - y1, z = z2 - z1, then the equation for the plane can be written in differential form as a x + b y + c z = 0. This equation is in window coordinates, so each scanline corresponds to a line of constant y, and y = 0 as we move across a scanline. On a scanline, we increase x in unit steps, corresponding to moving one pixel in the frame buffer, and x is constant. Thus, as we move from point to point across a scanline, z =- a c x. This value is a constant that needs to be computed only once for each polygon. Although the worst- case performance of an image- space algorithm is proportional to the number of primitives, the performance of the z- buffer algorithm is proportional to the number of fragments generated by rasterization, and this depends on the area of the rasterized polygons.

7.11.6 Scan Conversion with z-Buffer

a. Explain Figure 7.48

ANSWER:

(Angel pp. 365)



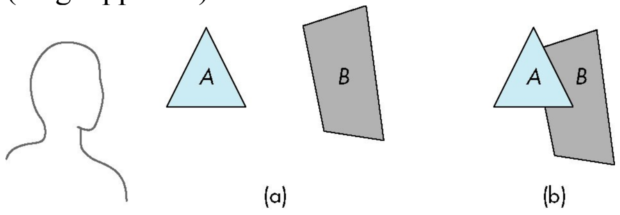
7.11.6 Scan Conversion with the z- Buffer We already have presented most of the essentials of polygon rasterization. In Section 7.10.1, we discussed the odd even and winding tests for determining whether a point is inside a polygon. In Chapter 6, we learned to shade polygons by interpolation. Here we have only to put together the pieces and to consider efficiency. Suppose that we follow the pipeline once more, concentrating on what happens to a single polygon. The vertices and normals pass through the geometric transformations one at a time. The vertices must be assembled into a polygon before the clipping stage. If our polygon is not clipped out, its vertices and normals can be passed on for shading and hidden- surface removal. At this point, although projection normalization has taken place, we still have depth information. If we wish to use an interpolative shading method, we can compute the lighting at each vertex. Three tasks remain: computation of the final orthographic projection, hidden-surface removal, and shading. Careful use of the z- buffer algorithm can accomplish

7.11.7 Depth Sort and the Painter's Algorithm

a. Explain Figure 7.51

ANSWER:

(Angel pp. 367)



7.11.7 Depth Sort and the Painters Algorithm Although image- space methods are dominant in hardware due to the efficiency and ease of implementation of the z- buffer algorithm, often object- space methods are used within the application to lower the polygon count. Depth sort is a direct implementation of the object- space approach to hidden- surface removal. We present the algorithm for a scene composed of planar polygons; extensions to other classes of objects are possible. Depth sort is a variant of an even simpler algorithm known as the painters algorithm. Suppose that we have a collection of polygons that is sorted based on how far from the viewer the polygons are. For the example illustrated in Figure 7.51(a), we have two polygons. To a viewer, they appear as shown in Figure 7.51(b), with the polygon in front partially obscuring the other. To render the scene correctly, we could find the part of the rear polygon that is visible, and could render that part into the frame buffer a calculation that requires clipping one polygon against the other. Or we could use another approach, analogous to the way a painter might render the scene. She probably would paint the rear polygon in its entirety, and then the front polygon, painting over the part of the rear polygon not visible to the viewer in the process. Both polygons would be rendered completely, with the hidden- surface removal being done as a consequence of the back- to- front rendering of the polygons. 9 The two questions related to this algorithm are how to do the sort, and what to do if polygons overlap. Depth sort addresses both, although in many applications more efficiencies can be found (see, for example, Exercise 7.10).

b. Explain Figure 7.52

ANSWER:

(Angel pp. 367)

Suppose that we have already computed the extent of each polygon. The next step of depth sort is to order all the polygons by how far away from the viewer their maximum z-value is. This step gives the algorithm the name depth sort. Suppose that the order is as shown in Figure 7.52, which depicts the z extents of the polygons after the sort. If the minimum depth the z- value of a given polygon is greater than the maximum depth of the polygon behind the one of interest, we can paint the polygons back to front and we are done. For example, in Figure 7.52, polygon A is behind all FIGURE 7.54 Polygons with overlapping extents the other polygons and can be painted first. However, the others cannot be painted based solely on the z- extents. If the z- extents of two polygons overlap, we still may be able to find an order to paint (render) the polygons individually and yield the correct image. The depth- sort algorithm runs a number of increasingly more difficult tests, attempting to find such an ordering. Consider a pair of polygons whose z- extents overlap. The simplest test is to check their x and y- extents (Figure 7.53). If either of the x or y extents do not overlap, 10 neither polygon can obscure the other, and they can be painted in either order. Even if these tests fail, it may still be possible to find an order in which we can paint the polygons individually. Figure 7.54 shows such a case. All the vertices of one polygon lie on the same side of the plane determined by the other. We can process the vertices (see Exercise 7.12) of the two polygons to determine whether this case exists.

B. (100 pts) Visual Studio 2008 C++ Project

B1. Create Visual Studio 2008 C++, Empty Project, Homework7:

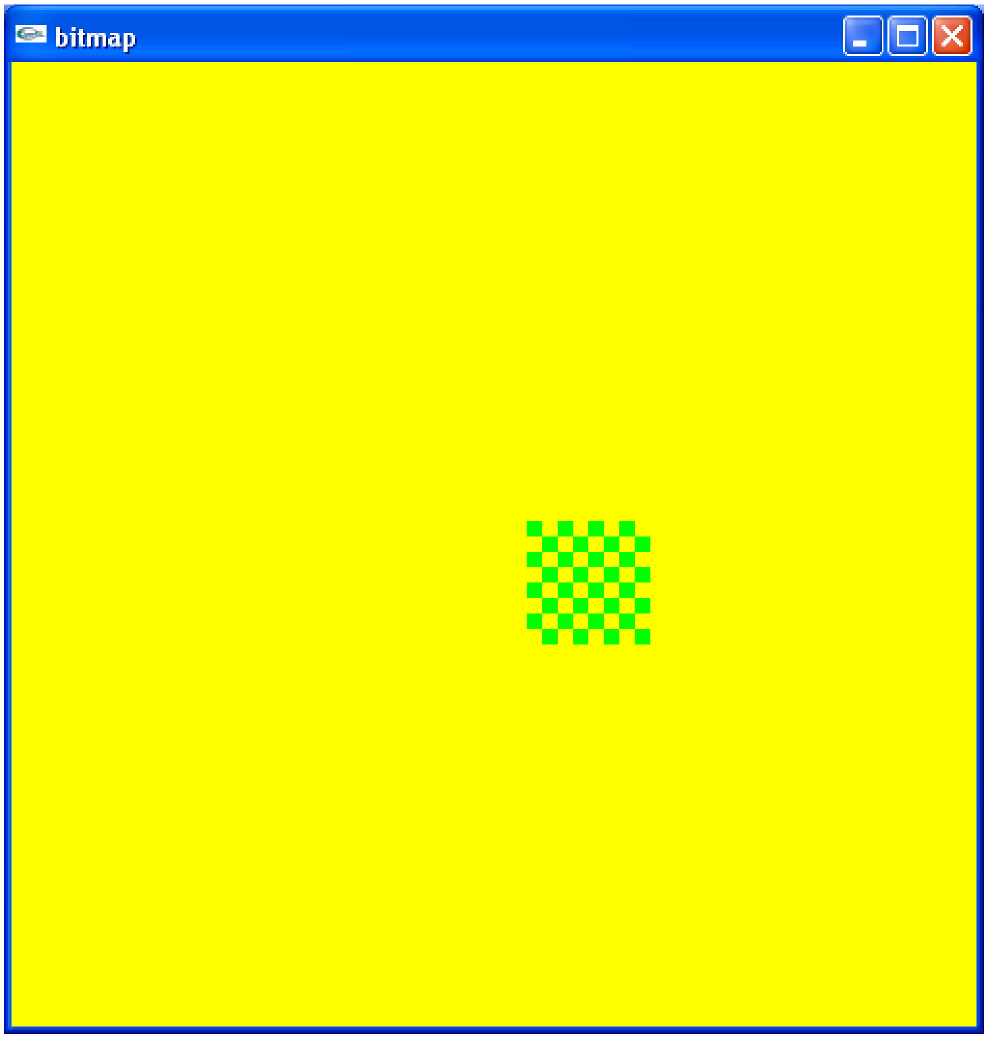
```
//bitmap.c

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

GLubyte wb[2]={0x00,0xff};
GLubyte check[512];
int ww;

void init()
{
   glClearColor(1.0, 1.0, 0.0, 1.0);
}

void display()
{
      glClear(GL_COLOR_BUFFER_BIT);
      glColor3f(1.0, 0.0, 0.0);
      glRasterPos2f(0.0, 0.0);
      glBitmap(64, 64, 0.0, 0.0, 0.0, 0.0, check);
      glFlush();
}

void reshape(int w, int h)
{
      glViewport(0, 0, w, h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      gluOrtho2D (0.0,  (GLfloat) w, 0.0, (GLfloat) h);
      ww=w;
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
}

void mouse(int btn, int state, int x, int y)
{

      if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
            {
            glBitmap(64, 64, 0.0, 0.0, 0.0, 0.0, check);
            glRasterPos2i(x, ww-y);
            glBitmap(64, 64, 0.0, 0.0, 0.0, 0.0, check);
            glFlush();
            }
}


int main(int argc, char** argv)
{
```

```
/* Initialize mode and open a window in upper left corner of screen */
/* Window title is name of program (arg[0]) */

       int i,j;
       for(i=0;i<64;i++) for(j=0;j<8;j++) check[i*8+j] = wb[(i/8+j)%2];
       glutInit(&argc,argv);
       glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
       glutInitWindowSize(500,500);
       glutInitWindowPosition(0,0);
       glutCreateWindow("bitmap");
       glutDisplayFunc(display);
       glutReshapeFunc(reshape);
       glutMouseFunc(mouse);
       init();
       glEnable(GL_COLOR_LOGIC_OP);
       glLogicOp(GL_XOR);
       glutMainLoop();

}
```

Build and run this Project: Insert a screenshot of your output. Explain how the `glRasterPos2f(0.0, 0.0);`

`glBitmap(64, 64, 0.0, 0.0, 0.0, 0.0, check);` work.

ANSWER:


bitmap